**Report**

**On**

**“Curse of Dimensionality**

**& Gradient Descent”**

**As a part of Course**

**Foundations of Data Science (CS F320)**

**By-**

**Pratik Kakade 2017A7PS0086P**

**Nevin Thomas 2017A7PS1175P**



**Birla Institute of Technology and Science, Pilani**

**Sem II, 2019-20**

**Supervised by: Dr. Navneet Goyal**

**Objective-**

The objective of this report is to understand :

1. problems related to high dimensional data and their solutions

2. gradient descent optimization technique and their solutions

**Part -1: Curse of Dimensionality**

**Introduction-**

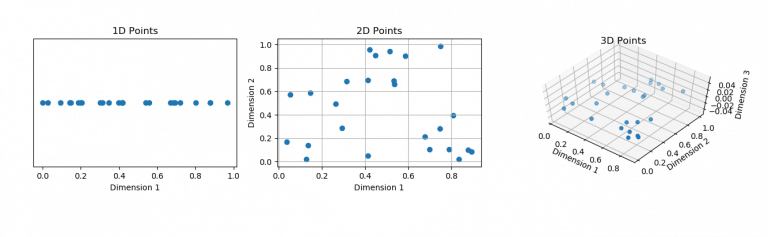
Dimensionality in data science refers to the number of attributes a dataset has. Data science is a field that often engages in analysis of structured and unstructured high dimensional data to infer patterns and insights through automated construction of models. Intuitively, one would think that the more the data to analyze the more better and accurate the conclusions the model could draw from them. However, as the dimensions of the data begin to increase, a several number of problems arise due to an unfortunate phenomenon known as the curse of dimensionality.

The main culprit behind the curse of dimensionality is that as the number of dimensions increase, the volume on which the model must be trained increases exponentially and the distribution or density of the data points through space becomes sparse, assuming the number is kept the same. This makes generalization difficult for the model as there is a larger region of uncertainty.

**Problems caused by curse of dimensionality:**

1. **Data sparsity and overfitting**

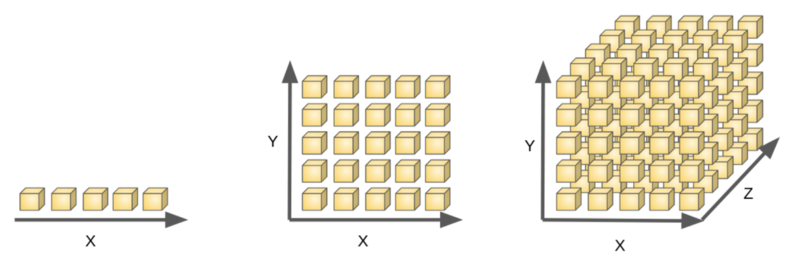
The below illustrated graphs contain 25 randomly sampled training points in the range [0,1]. We can observe that as the number of dimensions increases from left to right, the spaces between points starts increasing and the density of the points decreases. This is as the total region covered increases exponentially, making the data sparse.



If we keep the number of training points the same across all 3 graphs, it would lead to a problem called overfitting of the model in higher dimensions. Overfitting is when the model just memorizes the outcome for a set of points instead of generalizing a relationship over the entire training space. To ensure that the model maintains an accurate representation of the space, we would have to increase the number of points sampled exponentially as well. So, if the model needs 25 points in 1D for training, it would need 625 in 2D space and 15,625 in 3D space.

1. **Exponential increase in computations**

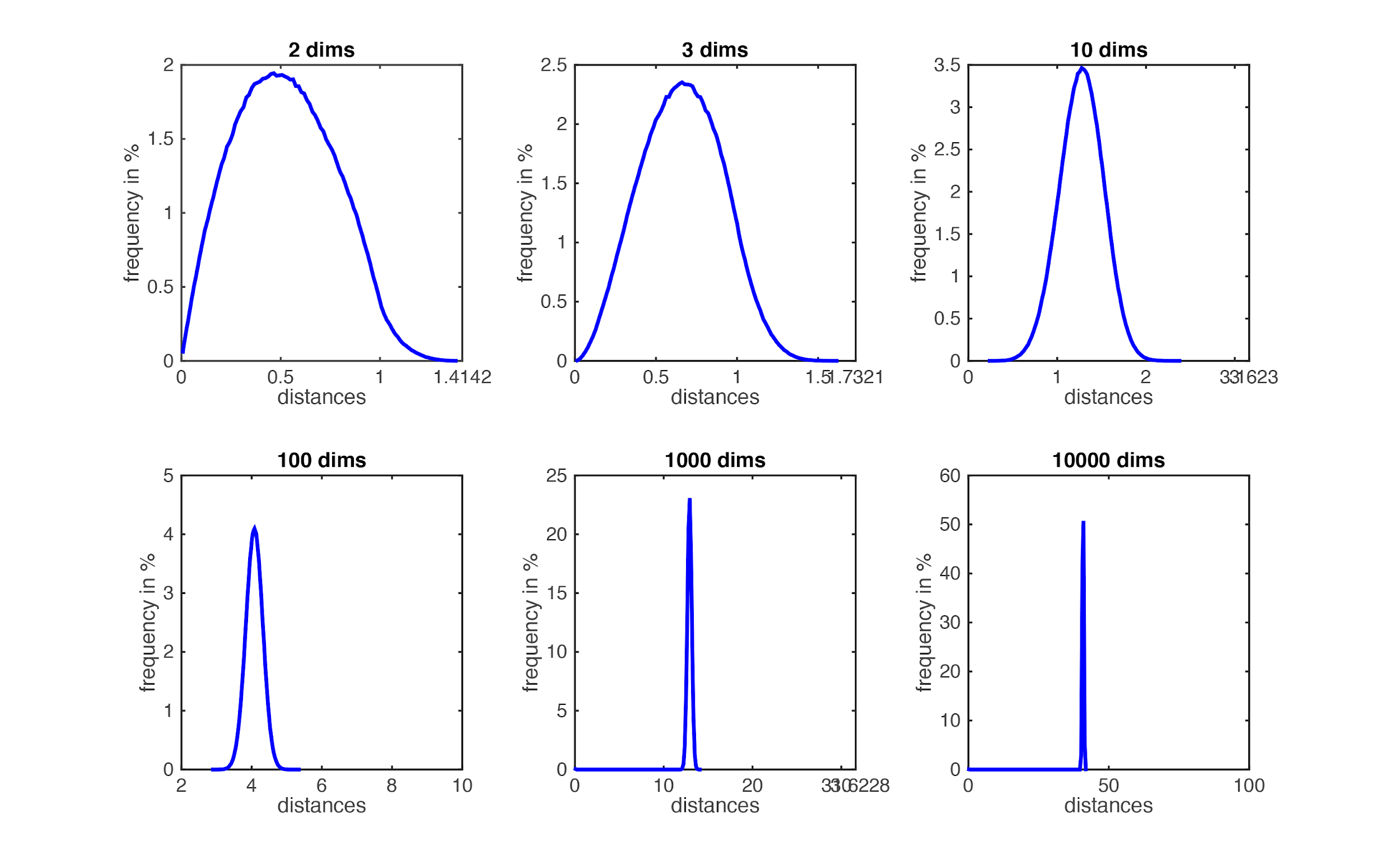
Even if we do increase the data exponentially with increase in dimensions as stated above, a new problem arises. Working with data becomes more and more demanding as the number of dimensions increases. To put it in a simple illustration, imagine that the points in n dimensional space were represented by a box and that one particular box held a key.



As illustrated in the figure, in n=1 you would have to search 5 boxes, in n=2 25 boxes and in n=3 125 boxes. As the number of dimensions grows the boxes to be searched increase exponentially. Or put in another way, the computational workload increases exponentially with increase in dimensions. With extremely high dimensional data, it is not easy to sample the entire region of feature space.

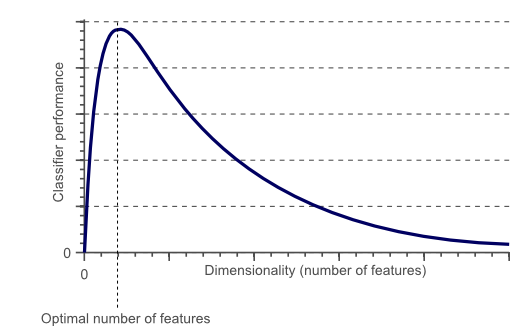
1. **Concentration of Lp norms**

Concentration of Lp norms refers to their inability as distance functions to distinguish points. We use Lp norms as a measure of closeness between two points. They play a critical role in almost every algorithm from classification to clustering to similarity searches. But as the dimension increases the distances between data points starts increasing until a point where all data points are almost equidistant from one another.



The above figure shows the distribution in pairwise distances as dimensionality increases. The bell shaped curve becomes more and more concentrated until it converges at infinity.

1. **Hubness**

Hubness is an aspect of the CoD arising from the concentration of Lp norms. Hubs are special points that occur often among the nearest neighbors of other objects. Conversely, some points may be classified as anti hubs, which are rarely or never nearest neighbors to other points. Many machine learning algorithms that rely on nearest neighbor search and some form of measuring distances, break down due to this reason. As the dimensionality of the dataset increases, only the hubs start to appear as the nearest neighbours and other points are disregarded. An example is the kNN classification algorithm. The algorithm classifies a point according to the majority class amongst its k nearest neighbours, the nearest neighbours being calculated by the L2 norm or Euclidean distance. But as dimensions increase, its accuracy reaches a maxima and then begins to drop drastically. The following graph demonstrates its variation in performance with dimensionality.

**Solutions to Curse of Dimensionality**

1. **Dataset size** : Ensure that the dataset is reasonably large enough.
2. **Dimensionality Reduction :**

It is not always feasible to have a large enough dataset proportionate to the number of dimensions. Also it is often the case that there are unnecessary or redundant features present in the dataset.Thus, the most preferred route to overcome the curse of dimensionality is dimensionality reduction. Dimensionality reduction is the process of reducing the information or features present in the dataset by filtering out the less significant data. It can be of two types - feature selection and feature extraction. The process of feature selection involves selecting a subset of the features present in the dataset while feature extraction involves creating new features by projecting data from higher dimensions to lower dimensional space. Another way to classify dimensionality reduction approaches can be supervised and unsupervised. Formally speaking, given the data matrix and label vector , where represents total number of data points and the number of dimensions, dimensionality reduction aims to create the best possible representation of the data matrix as where . While both approaches result in lower dimensional data, the process of supervised dimensionality reduction takes into account the label information to ensure that the dimension reduction is a direct aid to the process of model building.

**Solution of CoD implementation details**

Two datasets were used:Wine dataset and Madelon dataset. The code makes use of 4 different dimensionality reduction techniques, 2 unsupervised: PCA and SVD, and 2 supervised::Fisher's Linear discriminant for multiclass and Partial Least Squares Regression. The datasets are then compared using a linear SVM classifier.

Since both datasets have balanced classes, accuracy is judged to be a suitable enough performance metric.

The wine dataset was reduced from 13 dimensions to 10 to gain the following results.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| WINE  DATASET | Original dataset | FLD dataset | PCA dataset | SVD dataset | PLS dataset |
| Accuracy | 93.65079 | 98.46154 % | 95.38462% | 93.65079% | 93.84615% |

Only the training set of the madelon dataset was considered due to unavailability of test labels. The madelon dataset being part of a NIPS challenge had low accuracy due to its complexity and high non linearity. Nevertheless, it had an increase in accuracy by application of dimensionality reduction.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MADELON  DATASET | Original dataset | SVD  (20) | PCA  (40) | PCA+FLD  (15) | PCA+PLS  (15) |
| Accuracy | 52.49169% | 59.46844% | 60.82148% | 64.48% | 64.8% |

Note: Due to extremely high dimensionality(501) of the dataset, PCA was applied before applying FLD and PLS to reduce computation.

For both datasets, it is observed that dimensionality reduction is helpful in overcoming the curse of dimensionality.

**Part 2- Gradient Descent**

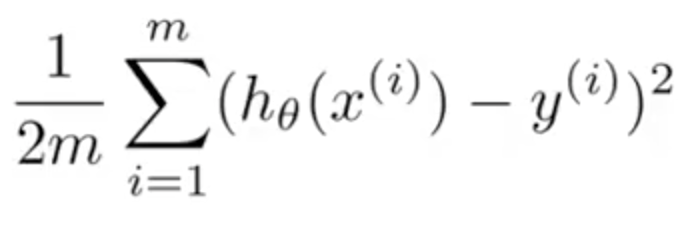
**Introduction**

A model often used in Machine Learning is linear regression- used for its simplistic nature and handling of multiple variables. This makes use of the formula:

Where: **f(X) is the output variable, and X1,X2….Xn****are the input variables, and 𝞱1,𝞱2,...𝞱n are the *function* *parameters.***

The goal of training the model is to get optimal function parameters such that the predicted output [**f(X)**] is as close to the actual values of the output variables[**y**].

In other words, we have to minimise the error function(or cost function):



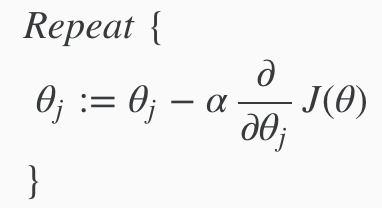
**Where m is the size of the dataset**

This is where gradient descent comes in. **Gradient descent** is an **optimization** algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. Here, we use gradient descent to update the parameters of our model and reduce the value of the given error equation.

**Method**

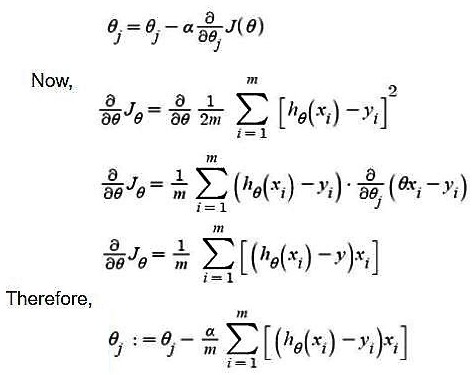
The method for gradient descent is as follows:

For every parameter 𝝷i in the function h(X):

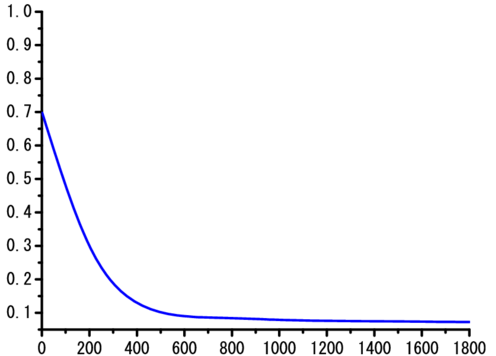
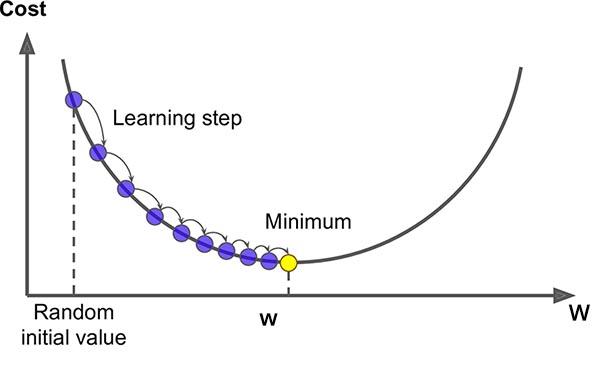


Where J(𝝷) is the *error function*

In other words:



To find a local minimum of a function using gradient descent, we take steps proportional to the negative of the gradient (or slope) of the function at the current point. As seen in the pictures below, the learning steps taken are big at first, but as the slope decreases, the jumps become lesser, till the slope becomes zero. This way, we are guaranteed to hit the minima of the cost function, which is exactly what we need to do for linear regression.

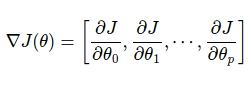


The term **𝛼** in the above equations refers to the ***learning rate***- it is the factor that defines how quickly we move towards the minima. With a high learning rate, we can cover more ground each step, but we risk overshooting the lowest point since the slope is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A low learning rate is more precise, but calculating the gradient is time-consuming, so it will take us a very long time to get to the bottom.

**Method of Gradient Descent Used In This Project**

In this project, rather than using the iterative approach as shown above, since the language R is very good at matrix multiplications, we have decided to use a matrix based approach,as shown:

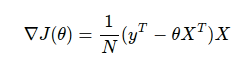
We have the gradient as :



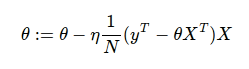
Where 𝞱 is an *1xn* matrix- of the parameters 𝞱0,𝞱1,𝞱2….𝞱n , n = number of attributes

y and X are the matrices of the output and input datasets- of dimensions *mx1* and *mxn* respectively.

Now, applying this to the datasets(according to previous formulae), we get the following:



Therefore, we iterate till convergence:



Advantages of Gradient descent

1. It is guaranteed to converge to the global minimum for convex error surfaces- thus guaranteeing the minimum possible error, in this case.

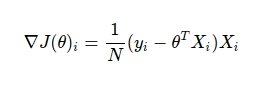
Disadvantages of gradient descent

1. If the learning rate is bigger, then convergence may not happen; we may keep on jumping past the minima. If it is very large, it may even “explode” the function and cause deviations from the minima.
2. Gradient descent is not able to find a global minima for non-convex functions; it can only identify a local one. It is also not possible to find minima of non-differentiable functions.
3. In every iteration, parameters are only updated only after an entire pass of all items in the dataset. For example, if there are 10000 instances in the data, the function will first calculate the error for 10000 samples, and then only update the parameters. This leads to rather slow convergence of parameters, since updation takes place only after a lengthy computation. The computation itself may be too computationally expensive, as the need to store large intermediate vectors in the calculation slows down the computation process.

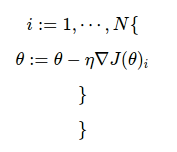
**Stochastic Gradient Descent**

As seen in the previous paragraph, convergence was quite slow due to only updating the parameters after computation over all the data points. In order to increase the speed of convergence, we use another variant of gradient descent, called *Stochastic Gradient Descent.* In this method one training sample is passed through the neural network at a time and the parameters are updated with the computed gradient. So, at a time a single training sample is passed through the network and its corresponding loss is computed. The parameters of all the layers of the network are updated after every training sample. For example, if the training set contains 10000 samples, then the parameters are updated 10000 times i.e. one time after every individual example is passed through the network.

In other words, for each row i in the dataset:



Then, the parameters are updated as follows:



***Where N is the size of the dataset***

Advantages:

1. It is easier to fit into memory due to a single training sample being processed by the network
2. It is computationally fast as only one sample is processed at a time
3. For larger datasets it can converge faster as it causes updates to the parameters more frequently
4. Due to frequent updates the steps taken towards the minima of the loss function have oscillations which can help getting out of local minimums of the loss function (in case the computed position turns out to be the local minimum)

Disadvantages:

1. Due to frequent updates the steps taken towards the minima are very noisy. This can often lead the gradient descent into other directions.
2. Also, due to noisy steps it may take longer to achieve convergence to the minima of the loss function
3. Frequent updates are computationally expensive due to using all resources for processing one training sample at a time
4. We lose the advantage of vectorized operations as it deals with only a single example at a time

**Mini-Batch Gradient Descent**

This is a variant of Gradient Descent that can be seen as a combination of the previous variants. The training set is divided into multiple groups called *batches*. Each batch has a number of training samples in it. At a time a single batch is passed through the network which computes the loss of every sample in the batch and uses their average to update the parameters of the neural network. For example, say the training set has 10000 training examples; which is divided into 50 batches with each batch containing 20 training examples.This means that the function will be iterated over 50 times (number of batches).

This is seen as the most practical solution for gradient descent as it combines the advantages of normal gradient descent and stochastic gradient descent:

1. Easily fits in the memory
2. It is computationally efficient- we can now benefit from vectorization as well
3. If stuck in local minimums, some noisy steps may lead the way out of them
4. Average of the training samples produces stable error gradients and convergence

This is the method used most in practice, with batch sizes typically small compared to the actual size of the dataset. In this project, we have chosen batch size to be 64.

**Implementation of Gradient Descent and Its Variants**

Here, we have used two datasets: the red-white wine quality dataset, and the life expectancy dataset from WHO.

*Preprocessing:*

1. The caret package was imported to help make data partitions and make dummy variables for categorical variables.
2. Any missing data(relatively few) was replaced with the average of the column.
3. All data has been scaled in order to make calculations simpler.
4. Outliers have been removed.

*Data Results:*

The three methods used are normal gradient descent, stochastic gradient descent and mini- batch gradient descent, abbreviated as GD,SGD and SGD\_MiniBatch respectively. All results seen below can be seen in a little more detailed fashion in the folder: **/Gradient\_Descent/experiment-results.** Here, the naming convention is as follows:

MethodUsedAbbr\_learningRateUsed.txt.

When a user runs the program now, the results are saved to the folder **/Gradient\_Descent/outputs** in the same format.

Now, we’ve put an upper limit on the number of iterations to 100,000. We’ve also set batch size to 64. These are the results of the experimentation based on tweaking the value of learning rate. Here, we note that the learning rate must be very small( < 0.0001) because the data values involved are also quite small.

***Learning rate 𝛼 = 0.0001, on Convergence - Life Expectancy data***

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***Number of iterations taken*** | ***Time taken(s)*** | ***Loss function*** |
| ***GD*** | ***18999*** | ***36.9*** | ***0.288361958441088*** |
| ***SGD*** | ***30*** | ***10.13*** | ***0.286313193337286*** |
| ***SGD-MiniBatch*** | ***900*** | ***8.87*** | ***0.298978052316772*** |

***Learning rate 𝛼 = 0.00005, on Convergence - Life Expectancy data***

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***Number of iterations taken*** | ***Time taken(s)*** | ***Loss function*** |
| ***GD*** | ***37999*** | ***97.47*** | ***0.297938910115163*** |
| ***SGD*** | ***50*** | ***23.85*** | ***0.282430351875748*** |
| ***SGD-MiniBatch*** | ***1400*** | ***14.79*** | ***0.295063050941678*** |

***Learning rate 𝛼 = 0.0001, on Convergence - Red-White Wine data***

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Number of iterations taken** | **Time taken(s)** | **Loss function** |
| **GD** | **26889** | **78.17** | **0.610804268140224** |
| **SGD** | **75** | **270.828** | **0.59845878098503** |
| **SGD-MiniBatch** | **600** | **13.1** | **0.594091256363894** |

***Learning rate 𝛼 = 0.00005, on Convergence - Red-White Wine data***

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Number of iterations taken** | **Time taken(s)** | **Loss function** |
| **GD** | **54341** | **166.11** | **0.595187724761217** |
| **SGD** | **51** | **135.96** | **0.599753590937894** |
| **SGD-MiniBatch** | **1900** | **72.1** | **0.600266161480045** |

From the above results, we can see that all 3 of the methods are effective in converging to the minima - the loss function is comparable for the 3. However, a clear comparison can be made in terms of the amount of time and number of iterations needed to converge. We can see that SGD- MiniBatch on average converges the fastest(in terms of time taken), while SGD takes the least number of iterations.

**Conclusion**

In this project, we described the Curse of Dimensionality and the problems associated with it. We then implemented solutions for it, using the methods of PCA,SVD, Fisher's Linear discriminant for multiclass and Partial Least Squares.

We also described the different variants of Gradient Descent and their strengths/weaknesses. We then implemented these types and compared their results.